# LLVM parallel intermediate representation: design and evaluation using OpenSHMEM communications

**Conference Paper** · January 2015

DOI: 10.1145/2833157.2833158



**5 authors**, including:

Dounia Khaldi
Stony Brook University
**17** PUBLICATIONS **19** CITATIONS
SEE PROFILE

Pierre Jouvelot
MINES ParisTech
**85** PUBLICATIONS **1,546** CITATIONS
SEE PROFILE

François Irigoin
MINES ParisTech
**137** PUBLICATIONS **1,688** CITATIONS
SEE PROFILE

Corinne Ancourt
MINES ParisTech
**42** PUBLICATIONS **652** CITATIONS
SEE PROFILE

# LLVM Parallel Intermediate Representation: Design and Evaluation using OpenSHMEM Communications *

Dounia Khaldi
Department of Computer Science
University of Houston
Houston, Texas 77004
dkhaldi@uh.edu

Pierre Jouvelot
MINES ParisTech
PSL Research University
France
pierre.jouvelot@mines-paristech.fr

François Irigoin
MINES ParisTech
PSL Research University
France
francois.irigoin@mines-paristech.fr

Corinne Ancourt
MINES ParisTech
PSL Research University
France
corinne.ancourt@mines-paristech.fr

Barbara Chapman
Department of Computer Science
University of Houston
Houston, Texas 77004
chapman@cs.uh.edu

## ABSTRACT

We extend the LLVM intermediate representation (IR) to make it a parallel IR (LLVM PIR), which is a necessary step for introducing simple and generic parallel code optimization into LLVM. LLVM is a modular compiler that can be efficiently and easily used for static analysis, static and dynamic compilation, optimization, and code generation. Being increasingly used to address high-performance computing abstractions and hardware, LLVM will benefit from the ability to handle parallel constructs at the IR level. We use SPIRE, an incremental methodology for designing the intermediate representations of compilers that target parallel programming languages, to design LLVM PIR.

Languages and libraries based on the Partitioned Global Address Space (PGAS) programming model have emerged in recent years with a focus on addressing the programming challenges for scalable parallel systems. Among these, OpenSHMEM is a library that is the culmination of a standardization effort among many implementers and users of SHMEM; it provides a means to develop light-weight, portable, scalable applications based on the PGAS programming model. As a use case for validating our LLVM PIR proposal, we show how OpenSHMEM one-sided communications can be optimized via our implementation of PIR into the LLVM compiler; we illustrate two important optimizations for such operations using loop tiling and communication vectorization.

## Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers


LLVM-HPC2015, November 15-20, 2015, Austin, TX, USA
 ISBN 978-1-4503-4005-2/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2833157.2833158

## Keywords

Parallel language intermediate representation, LLVM, PGAS, OpenSHMEM, SPIRE

## 1. INTRODUCTION

The growing importance of parallel computers and the search for an efficient programming model led, and is still leading, to a proliferation of parallel programming languages. To adapt to this evolution, existing compilers such as LLVM [14] need to be equipped with internal intermediate representations (IR) for parallel constructs. The choice of a proper parallel IR (PIR) is of key importance, since the efficiency and power of the transformations and optimizations are closely related to the selection of a proper program representation paradigm.

A common technique for introducing parallel constructs in sequential IRs, adopted by LLVM to support OpenMP for example, is to encode them as "fake" function calls at the front-end phase (in Clang). Although simple, this ad-hoc solution prevents high-level reasoning about parallel languages while possibly introducing semantic inconsistencies in existing sequential program analyses. Another approach is advocated by SPIRE [13, 12], a sequential-to-parallel intermediate representation extension methodology that can be applied to existing compilers in a rather simple manner. Its core philosophy is to extend in a systematic manner the IRs found in the compilation frameworks of sequential languages. Avoiding the ad-hoc "fake" function call approach, SPIRE enables the leveraging of current compilation infrastructures for sequential languages to address both control and data parallel constructs while preserving as much as possible the correctness of existing analyses for sequential code. SPIRE is based on only three key concepts for extension: (1) the parallel vs. sequential execution of groups of statements such as sequences, loops and general control-flow graphs, (2) the global synchronization characteristics of statements and the specification of finer grain synchronization via the notion of events and (3) the handling of data distribution for different memory models.

In this paper, we apply SPIRE to the IR of LLVM, a widespread SSA-based compilation infrastructure for sequential and parallel languages. This yields the Parallel IR, noted LLVM PIR, we included into the LLVM framework and are currently experimenting

with. Our design is the result of many trade-offs between generality and precision, abstraction and low-level concerns. On the one hand, and in particular when looking at the low-level features of the LLVM IR, one needs to be able to represent as many of the existing (and, hopefully, future) parallel constructs while minimizing the number of new concepts introduced in the parallel IR. Yet, keeping only a limited number of hardware-level notions in the IR, while good enough to deal with all parallel constructs, could entail convoluted rewritings of some high-level parallel flows.

To validate the benefits of our LLVM PIR proposal on actual parallel code, we describe the application, via our PIR-equipped LLVM compiler implementation, of two optimizations on OpenSHMEM microbenchmarks and exhibit their improved run-time performance. OpenSHMEM [4] is a library interface specification that is the culmination of a unification effort among many vendors and users in the SHMEM programming community. It is designed with a chief aim of performance, exploiting support for Remote Direct Memory Access (RMA) available in modern network interconnects and thereby allowing for highly efficient data transfers. On such operations, we show the positive impact of loop tiling and communication vectorization, two optimization techniques deployed within LLVM and easily specified on PIR-encoded structures.

The main contributions of this paper are:

- the design of LLVM PIR, a parallel intermediate representation to be used in the LLVM compilation framework, via the application of SPIRE, a simple, parallel intermediate representation extension methodology;
- the conversion of OpenSHMEM one-sided communication calls into LLVM volatile load/store operations – we use *metadata* to express remote processes and *volatile* annotations to avoid unsafe optimizations;
- an implementation of LLVM PIR into the LLVM compiler, to validate SPIRE for the parsing of the parallel library OpenSHMEM and enabling important optimizations for one-sided communications using loop tiling and communication vectorization.

After this introduction, we describe the LLVM compilation framework and its sequential IR, in Section 2, and provide a quick overview of OpenSHMEM. Our parallel extension proposal, LLVM PIR, is introduced in Section 3. Section 4 illustrates and discusses performance results of the application of LLVM PIR on OpenSHMEM one-sided communications. We survey existing parallel IRs in Section 5. We discuss future work and conclude in Section 6.

# 2. BACKGROUND

In this section, we present the LLVM compiler, its intermediate representation and OpenSHMEM.

## 2.1 LLVM (Sequential) IR

In this work, we provide a parallel version of the IR of widely-used LLVM [14] (Low-Level Virtual Machine). LLVM is an open-source compilation framework that uses an intermediate representation in Static Single Assignment (SSA) [5] form. LLVM includes multiple tools, especially Polly [7] for optimizing memory accesses. Polly [7] is a high-level loop and data-locality optimizer for LLVM. It uses the polyhedral model to analyze and optimize the memory access patterns of a program. Polly also performs classical loop transformations, especially tiling and loop fusion, to improve data locality.

One motivation for using LLVM is that it has been widely used in both academia and industry. Another interesting feature of LLVM IR is that it sports a graph approach; each function is structured in LLVM as a control-flow graph (CFG). Figure 1 provides the definition of a significant subset of the sequential LLVM IR described in [15]. It is specified using Newgen [11], a Domain Specific Language for the definition of set equations from which a dedicated API is automatically generated to manipulate (creation, access, IO operations...) data structures implementing these set elements.

The main components of LLVM IR are summarized in Figure 1.

- A function is a list of basic blocks, which are portions of code with one entry and one exit points; the "$\star$" symbol introduces lists.
- A basic block has an entry label, a list of $\phi$ nodes and a list of instructions, and ends with a terminator instruction. Instructions are included within basic blocks, which are members of a cartesian product set (noted `x`). Load and store instructions are members of the disjoint union set `instruction` (noted +).
- $\phi$ nodes, which are the key elements of SSA, are used to merge the values coming from multiple basic blocks. A $\phi$ node is an assignment (represented here as a call expression) that takes as arguments an `identifier` and a list of pairs (value, label); it assigns to the identifier the value corresponding to the label of the block preceding the current one at run time.
- Every basic block ends with a terminator, which is a control flow-altering instruction that specifies which block to execute after completion of the current one.

```
function    = blocks:block*;
block       = label:identifier x phi_nodes:phi* x
    instructions:instruction* x terminator;
phi         = call;
instruction = load + store + call;
load        = identifier;
store       = name:identifier + value:expression;
terminator  = cond_br + uncond_br + return;
cond_br     = value:identifier x
    label_true:identifier x label_false:identifier;
uncond_br   = label:identifier;
return      = value:identifier;
```

Figure 1: Simplified definitions of the LLVM IR

An example of the LLVM encoding of a simple "for" loop in three basic blocks is provided in Figure 2. In the assignment to `%sum.0`, which uses a $\phi$ node, `sum` is `42` if it is the first time we enter the `bb1` block (from `entry`) or the updated `sum` otherwise (from the branch `bb`).

## 2.2 OpenSHMEM

OpenSHMEM offers many features for supporting Partitioned Global Address Space (PGAS)-based applications, such as (1) management of remotely-accessible variables, (2) one-sided data communication, (3) barrier and point-to-point synchronizations, (4) atomic

```
sum = 42;
for(i=0;i<10;i++)
  sum = sum + 2;
```

```
entry:
 ...
 br label %bb1
bb:       ; preds = %bb1
 %0 = add nsw i32 %sum.0, 2
 %1 = add nsw i32 %i.0, 1
 br label %bb1
bb1:      ; preds = %bb, %entry
 %sum.0 =
   phi i32 [42,%entry], [%0,%bb]
 %i.0 =
   phi i32 [0,%entry], [%1,%bb]
 %2 = icmp sle i32 %i.0, 10
 br i1 %2, label %bb, label %bb2
bb2:      ; preds = %bb1
 ...
```

Figure 2: A loop in C and its LLVM intermediate representation

memory operations, and (5) collective operations such as reductions. OpenSHMEM programs follow an SPMD-like style of programming where all Processing Elements (PEs) are launched at the beginning of the program. The general objective of the OpenSHMEM library is to provide explicit control over data transfers with low-latency communications. Note that, in our experiments (see Section 4), we focus on optimizing one-sided communications of OpenSHMEM, namely `shmem_put` and `shmem_get`.

# 3. DESIGNING LLVM PARALLEL IR

In this section, we present the steps taken to extend the LLVM sequential IR to LLVM Parallel IR via the SPIRE transformation methodology.

## 3.1 SPIRE

SPIRE [12] (for Sequential to Parallel IR Extension Methodology) is a design framework dedicated to the introduction, in a systematic manner, of parallel concepts within sequential IRs. In [13], we defined the small-step, operational semantics of the SPIRE transformation process, to formally specify how its key parallel concepts are added to existing systems. Compiler writers following SPIRE guidelines can easily include parallel traits to their IR, enabling the generation of parallel code from sequential programs or the handling of explicitly parallel programming languages. The use of SPIRE does not intend to lead to minimal parallel IR extensions but to parallel IRs as seamlessly as possible integrable within actual IRs, while handling as many parallel programming constructs as possible. Its design point provides proper trade-offs between generality, expressibility and conciseness of representation. The main parallel constructs targeted by SPIRE are (1) launching task and data parallel computations, (2) performing synchronization, (3) introducing atomic sections and (4) transferring data according to various memory models.

To deal with parallel programming, SPIRE suggests to extend a given sequential IR with the ability to specify (1) the parallel execution of groups of statements, (2) the synchronization between statements and (3) data layout, i.e., how memory is modeled in a given parallel language. SPIRE is based on the introduction of key parallelism-related notions, collected in three groups: (1) execution, via the `parallel` and `reduced` constructs; (2) synchronization, via the `spawn`, `barrier`, `atomic` and event constructs; and (3) data distribution, via `send`, `recv` and `location` constructs. We will detail these constructs in the following subsections where we describe the application of SPIRE on the LLVM sequential IR to generate our LLVM PIR (Parallel IR) proposal.

## 3.2 Execution

The issue of parallel vs. sequential execution appears when dealing with groups of statements, which in our case corresponds to members of the `function` and `block` sets. Following SPIRE guidelines, an `execution` attribute is added to these sequential set definitions:

```
function'     =   function x execution;
block'        =   block x execution;
```

The primed sets `function'` and `block'` (implementing control parallelism) represent SPIREd-up sets for the LLVM parallel IR. Of course, the 'prime' notation is used here for pedagogical purpose only; in practice, an execution field is added in the existing IR representation. The definition of `execution` is straightforward:

```
execution  = sequential:unit +
         parallel:scheduling + reduced:unit;
scheduling = static:unit + dynamic:unit +
         speculative:unit + default:unit;
```

where `unit` denotes a set with one single element; this encodes a simple enumeration of cases for execution. A `parallel` (resp. `reduced`) execution attribute asks for all statements in a block to be all launched in parallel, in an implicit flat fork/join (resp. left-to-right, tree-like) fashion. LLVM PIR is an extendable framework; this can be shown with the generalization of the definition for the `parallel` case, where scheduling properties such as dynamic or speculative are introduced.

## 3.3 Synchronization

Synchronization behavior is a characteristic feature of the run-time properties of one statement with respect to other statements. SPIRE suggests to extend sequential intermediate representations by adding a synchronization attribute to the specification of statements. Applying these guidelines to LLVM IR yields:

```
block''       = block' x synchronization;
instruction' = instruction x synchronization;
```

Coordination by synchronization in parallel programs is often managed via coding patterns such as barriers, used for instance when a code fragment contains many phases of parallel execution where each phase should wait for the precedent ones to proceed. We define the `synchronization` set via high-level coordination characteristics useful for optimization purposes:

```
synchronization = none:unit +
            spawn:identifier + barrier:unit +
            atomic:identifier;
```

Assume $S$ is the instruction with a synchronization attribute.

- `none` specifies the default behavior, i.e., independent with respect to other statements, for $S$.

- `spawn` induces the creation of an asynchronous task $S$, while the value of the corresponding identifier is the user-chosen number of the thread that executes $S$. By convention, we say that `spawn` creates processes in the case of the message-passing and PGAS memory models, and threads, in case of the shared memory model. Since `spawn` impacts the flow of control of $S$ with respect to its surrounding instructions, we see `spawn` as a synchronization characteristic of $S$.
- `barrier` specifies that all the children threads spawned by the execution of $S$ are suspended before exiting until they are all finished.
- `atomic` predicates the execution of $S$ to the acquisition of a lock to ensure exclusive access; at any given time, $S$ can be executed by only one thread. Locks are logical memory addresses, represented here by a member of the LLVM IR `identifier` set.

## 3.4 Data Distribution

The ability of handling the various memory models used by parallel languages is an important issue when designing a generic intermediate representation. One currently considers there are three main parallel memory models: shared, message passing and, more recently, PGAS. This last model, which appears in languages such as CAF [18], Chapel [3], OpenSHMEM [4] and X10 [2], introduces various new notions such as `image`, `place` or `locale` to label portions of a logically-shared memory that processes may access, in addition to complex APIs for distributing data over these portions.

### *3.4.1 Memory Model*

SPIRE is able to handle all three memory models using specific memory information, namely private, shared and *PGAS*. Each process has its own private memory; the address of a shared variable refers, within each thread, to the same physical memory location; PGAS memory is distributed evenly among different processes. In this last case, e.g., in OpenSHMEM and CAF, all processes have their own view of PGAS memory.

To handle memory information, SPIRE extends the definition of the storage feature of identifiers by specifying where a given identifier is stored, i.e., within private, shared or PGAS memory. For LLVM IR, this leads to adding a `location` domain to the `identifier` domain:

```
identifier' = identifier x location;
location    = private:unit +
              shared:unit + pgas:unit;
```

We show below two examples related to memory in `pgas` location. First, in CAF, coarrays, which are data entities that can be directly referenced or defined by any image and may be a scalar or an array, are PGAS arrays. A coarray has codimensions, which specify the image to which it belongs. In the following example, the `dest` coarray of 20 elements will be visible and accessible remotely by all images.

```
integer(len=20) :: dest[*]
```

Our second example shows an OpenSHMEM statement allocating `dest`, which is also remotely accessible by all PEs.

```
dest = (int*)shmalloc(sizeof(int)*20);
```

### *3.4.2 One-Sided Memory Access*

In one-sided communications, only the source or destination process participates in asynchronous memory accesses, decoupling thus data transfer and synchronization. To account for the explicit distribution required by the PGAS memory model used in parallel languages such as CAF or libraries such as OpenSHMEM, SPIRE extends the traditional semantics of memory accesses and assignments. Since the information needed for specifying remote accesses is already present in the `location` domain of identifiers, there is no need to gather again this information. Put operations copy data from a local source memory area to a memory area of the remote target (`get` and `put` are dual operations). The following function call in OpenSHMEM:

```
shmem_int_put(dest, src, 20, pe);
```

can be represented in LLVM PIR, using here a C-like language notation, by:

```
for(i=0; i<20; i++)
 dest{pe}[i]= src[i];
```

where `location` of `dest` is `pgas`. The `identifier` domain is extended to handle expressions in the left hand side of assignments. Note that, in our case, at the code generation phase (after optimizations), this loop of assignments is transformed back to the OpenSHMEM library call `shmem_int_put(dest, src, 20, pe)`.

## 3.5 Intrinsics

In SPIRE, the goal is to design minimalist but general enough parallel IRs to handle as many parallel programming constructs as possible. For some constructs such as two-sided communications and point-to-point synchronization, SPIRE suggests to introduce intrinsic functions to handle them as in the following.

### *3.5.1 Two-Sided Memory Access*

In order to take into account the explicit distribution required by the message passing memory model used in parallel languages such as MPI, SPIRE introduces the `send` and `recv` blocking functions for implementing communication between processes:

- `void send(int dest, identifier buf)` transfers the value of identifier `buf` to the process numbered `dest`;
- `void recv(int source, identifier buf)`, in a dual fashion, receives in `buf` the value sent by the process numbered `source`.

Non-blocking communications can be implemented within the SPIRE methodology by using the above primitives within `spawn`ed statements. Also, broadcast collective communications, such as defined in MPI, can be seen as wrappers around `send` and `recv` operations. When the master process and receiver processes want to perform a broadcast function, then, if this process is the master, its broadcast operation is equivalent to a loop over receivers, with a call to `send` as body; otherwise (receiver), the broadcast is a `recv`

function. Note that, in our case, after the optimization phases of the LLVM compiler, at the backend stage, runtime library calls are generated to take advantage of the support for optimized lower-level communication functions provided via OpenSHMEM.

#### 3.5.2 Event API: Point-to-Point Synchronization

In parallel code, one usually distinguishes between two types of synchronization: (1) collective synchronization between threads using barriers, handled in LLVM PIR via the `synchronization` patterns above, and (2) point-to-point synchronization between participating threads. Handling point-to-point synchronization using decorations on abstract syntax trees or control-flow graphs is too constraining when one has to deal with a varying set of threads that may belong to different parallel parent nodes. Thus, SPIRE suggests to deal with this last class of coordination by introducing new values, of type `event`. Thus, to handle point-to-point synchronization, the underlying type system of LLVM IR is extended with a new basic type, namely `event`:

```
type′ = type + event:unit ;
```

Values of type `event` are counters, in a manner reminiscent of semaphores [6]. The programming interface for events is defined by the following functions:

- `event newEvent(int i)` creates an event, initialized with the integer `i` that specifies how many threads can execute `wait` on this event without being blocked;
- `void signal(event e)` increments by one the event value of `e`;
- `void wait(event e)` blocks the thread that calls it until the value of `e` is strictly greater than 0. When the thread is released, this value is decremented by one.

Note that the `void` return type can be replaced by `int` in practice, to enable the handling of error values, and that a `free` function may be needed in some languages.

Representing high-level point-to-point synchronization features such as phasers and futures using low-level functions on events constitutes a trade-off between generality and conciseness. One can represent any type of synchronization via the low-level interface using events: of course, this way, one may lose some of the expressiveness and precision of these high-level constructs such as the deadlock-freedom safety property of phasers [19]. Indeed, Habanero-Java parallel code avoids deadlock by imposing that the scope of each phaser is the immediately enclosing `finish` statement. However, the scope of the events is less restrictive; the user can put them wherever he wants, and this may lead to deadlocks.

# 4. EXPERIMENTAL EVALUATION

After summarizing LLVM PIR, we describe our implementation inside the LLVM compiler and the tests and optimizations enabled by our parallel IR proposal. Our goal is two-fold: (1) to show that our LLVM PIR proposal can be effectively used to encode parallel constructs, here using the PGAS memory model, and (2) to illustrate how the SPIRE approach is useful to leverage existing optimization passes present in the LLVM sequential compiler to parallel constructs.

## 4.1 LLVM PIR

Applying SPIRE to LLVM IR, as detailed in the previous section, yields the SPIREd parallel extension of the LLVM sequential IR; LLVM PIR is summarized in Figure 3.

- An `execution` attribute is added to `function` and `block`: a parallel basic block sees all its instructions launched in parallel (in a fork/join manner), while all the blocks of a parallel function are seen as parallel tasks to be executed concurrently.
- A `synchronization` attribute is added to `instruction`; hence, an instruction can be annotated with `spawn`, `barrier` or `atomic` synchronization attributes. When one wants to deal with a sequence of instructions, this sequence is first englobed in a `block` to which `synchronization` is added. Besides, as LLVM provides a set of intrinsic functions [14], SPIRE functions `newEvent`, `signal` and `wait` for handling point-to-point synchronization are added to this set.
- `send` and `recv` functions for handling data distribution are also seen as intrinsics. Moreover, `pgas` variables are introduced in LLVM PIR by enriching the `identifier` domain with `location` information. One-sided communications are represented by adding `expression` to `load` (RMA get) and `store` (RMA put) instructions.

```
function′     = function x execution;
block′        = block x execution x synchronization;
instruction′  = instruction x synchronization;
load′         = load x expression;
store′        = store x expression;
identifier′   = identifier x location;
type′         = type + event:unit;
```

Figure 3: SPIRE (LLVM IR), i.e., LLVM PIR

## 4.2 LLVM PIR Implementation

Since LLVM IR encodes code information using only low-level operations, our goal was to reuse existing low-level operations to implement the concepts presented in Figure 3.

As a use case for testing our LLVM PIR proposal, we looked in detail at the issue of OpenSHMEM one-sided communication optimization, via the middle-end optimization layer of LLVM. SPIRE suggests to represent the one-sided communication primitives of OpenSHMEM in LLVM IR by (remote) memory load/store operations. The goal of our implementation was to make minimal changes to LLVM IR. Thus, we used existing functionalities in LLVM IR to represent the RMAs of OpenSHMEM, namely load/store operations, metadata and volatile memory accesses. We implemented the `location` domain information as annotations on the IR nodes, using LLVM metadata, which is a simple string added to LLVM IR nodes.

LLVM provides the possibility to mark memory accesses as volatile. In this case, the LLVM optimizer must not change the number of volatile operations or change their order of execution relative to other volatile operations. In our implementation, we made use of this concept and we marked remote memory accesses (both read and write) as volatile to avoid unsafe optimizations, especially dead code elimination. However, safe optimizations such as loop unrolling can be applied by the LLVM optimizer.

This whole transformation process allows the middle-end optimization phases of LLVM and Polly to identify and optimize communications in OpenSHMEM, seen as "simple" load/store operations and not opaque function calls. We only had to extend the LLVM backend to generate the one-sided put/get calls `shmem_put` and `shmem_get` to be executed by the OpenSHMEM runtime. Note that at run time, two successive OpenSHMEM put/get operations may deliver data out of order unless a call to `shmem_fence` is introduced between the two calls.

## 4.3 One-Sided Communication Optimizations

We use as microbenchmark [1] a single OpenSHMEM statement, namely `shmem_int_put`, between a pair of processes, while varying the size of the transmitted array. Recall that such an instruction is encoded in LLVM PIR as a loop of remote write instructions. Figure 4 shows the LLVM PIR encoding of such a statement. We ran it on the Stampede supercomputing system at Texas Advanced Computing Center (TACC), under the OpenSHMEM implementation in MVAPICH2-X [10] version 2.0b. We compiled with LLVM-3.5.0, sporting the same version of Polly.

As a specific case study of the benefits induced by our use of LLVM PIR, we show the impact of two optimizations: loop tiling and communication vectorization. Loop tiling permits to overlap computation and communication, while communication vectorization reduces the number of communications.

### *4.3.1 Loop Tiling*

Loop tiling is a classical sequential optimization already available in Polly. We applied loop tiling on a SPIRE-encoded OpenSHMEM program; the idea is to break down communications into successive chunk transfers using loop tiling in order to achieve high throughput. Loop tiling was performed via the unmodified LLVM middle-end optimizer module (`opt`), operating on the loop of (remote) memory load/store operations. We relied upon Polly, in conjunction with `opt`, for carrying out the polyhedral analyses and optimizations required for loop tiling.

Figure 5 shows that `opt` extended with Polly was able to improve communication operations via tiling. This is particularly visible for large messages ($\geq$ 2048), since we chose a tile size of 2048 (`-polly-tile-sizes=2048`). The MVAPICH2-X implementation already provides optimizations for small and medium message sizes. Indeed, for such messages, an intermediate buffer is used to first copy the source data of the OpenSHMEM RMA call, thus making such calls non-blocking locally.

### *4.3.2 Communication Vectorization*

We performed automatic vectorization [9] of one-sided communications, which transforms a set of load/store operations automatically into bulk put/get communications in order to minimize the number of actual communications. As a simple example, the code for `N` one-element transfers (so-called *fine* transfer):

```
for(i=0; i<N; i++)
    shmem_int_put(dest[i],src[i],1,pe);
```

can be transformed in LLVM PIR to a bulk transfer:

```
dest{pe}[1:N]=src[1:N];
```

This assignment will be transformed again after middle-end optimizations to the run-time library call:

```
shmem_int_put(dest, src, N, pe);
```

We used `LoopIdiomRecognize`, which is an existing LLVM optimization that transforms simple loops of load/store operations into a non-loop format such as `memcpy`. In this work, we adapted this existing `LoopIdiomRecognize` LLVM pass to work on (remote) load/store operations that have the same RMA metadata (remote PE). Of course, put/get operations are generated here in lieu of `memcpy` intrinsics.

Figure 6 shows the impact of communication vectorization, which reduces both message startup time and latency, thanks to our modified `LoopIdiomRecognize` transformation.

Had we not used our SPIRE encoding, LLVM (and any other compiler) would not have been able to apply these two optimizations, since it would have considered blindly the put operation as a function call with no particular semantic.

# 5. RELATED WORK

In this section, we review different intermediate representations in existing parallel compilers and how they handle parallel programs. Also, we review specific works done in LLVM to support PGAS languages.

Syntactic approaches to parallelism expression use abstract syntax tree nodes, while adding specific parallel built-in functions. For instance, the IR of the implementation of OpenMP in GCC (GOMP) [17] extends its three-address representation, GIMPLE [16]. The OpenMP parallel directives are replaced by specific built-ins in low- and high-level GIMPLE, and additional nodes in high-level GIMPLE, such as the `__sync_fetch_and_add` built-in function for an atomic memory access addition. Similarly, Sarkar and Zhao introduce the high-level parallel IR HPIR [21] that decomposes Habanero-Java programs into region syntax trees, while maintaining additional data structures on the side: region control-flow graphs and region dictionaries. New syntax tree nodes are introduced: `AsyncRegionEntry` and `AsyncRegionExit` delimit tasks, while `FinishRegionEntry` and `FinishRegionExit` can be used in parallel sections. In this paper, the LLVM PIR designed along SPIRE guidelines borrows some of the ideas used in GOMP or HPIR, but frames them in more structured settings while trying to be more language-neutral. In particular, we try to minimize the number of additional built-in functions, which have the drawback of hiding the abstract high-level structure of parallelism and affecting compiler optimization passes.

The LLVM compiler supports OpenMP, but lowers all its pragmas at the front-end phase (in Clang) to runtime calls. In this work, we added specific support for the one-sided operations of OpenSHMEM in LLVM, via load/store constructs of LLVM IR. This makes it possible to apply seamlessly LLVM transformations such as loop tiling and communication vectorization to OpenSHMEM programs.

Finally, our work is among the current efforts to uniformly optimize PGAS programs using the LLVM infrastructure. There have been other works to optimize PGAS languages using the LLVM compiler, namely the Chapel and UPC compilers. A set of communica-

```
LoadStoreLoop:  ; preds = %LoadStoreLoop, %entry
  %lafee = phi i64 [ 0, %entry ], [ %nextvar, %LoadStoreLoop ]
  %addressSrc = getelementptr i32* getelementptr inbounds ([11 x i32]* @src, i32 0, i32 0), i64 %lafee
  %addressDst = getelementptr i32* getelementptr inbounds ([11 x i32]* @dest, i32 0, i32 0), i64 %lafee
  %RMA = load i32* %addressSrc
  store volatile i32 %RMA, i32* %addressDst, !PE !{i32 %7}
  %nextvar = add i64 %lafee, 1
  %cmptmp = icmp ult i64 %nextvar, %conv
  br i1 %cmptmp, label %LoadStoreLoop, label %shmem_RDMA_bb
```

Figure 4: LLVM PIR encoding of the OpenSHMEM statement `shmem_int_put(dest, src, N, pe)`

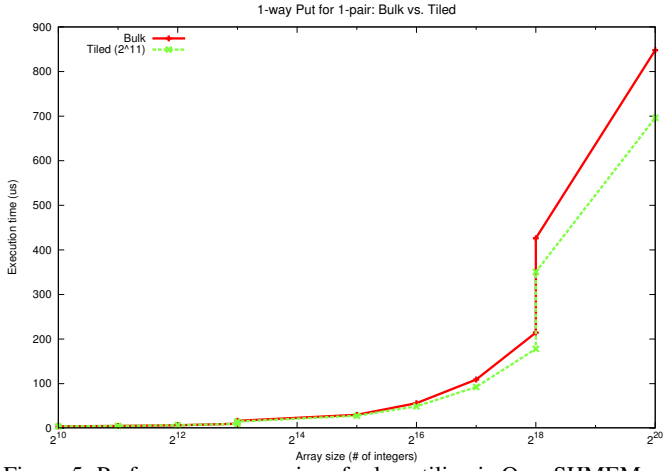


Figure 5: Performance comparison for loop tiling in OpenSHMEM on Stampede using 2 nodes

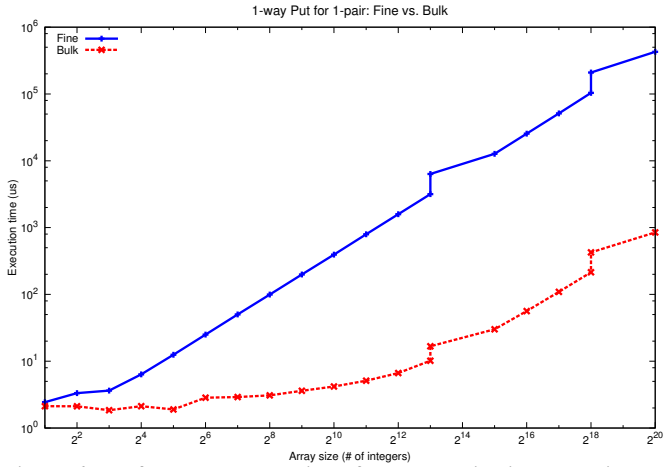


Figure 6: Performance comparison for communication vectorization in OpenSHMEM on Stampede using 2 nodes

tion optimizations in Chapel has been implemented in LLVM [8]. Communications in Chapel are implicit; they are simple assignments in the code. The Chapel compiler generates LLVM IR. The compiler/runtime have then to figure out if the access is local or remote. In this referenced paper, if an access is remote, the address space feature of LLVM is used with the value 100 to denote it. After that, loop invariant code motion optimization is applied to convert remote accesses to local accesses within a loop. Also, an implementation of UPC is available in LLVM [20]. In this work also, the address space feature of LLVM IR is used to represent shared variables. LLVM load and store IR instructions are used to express UPC shared memory accesses. However, in both works, the issue of how the LLVM optimizations are impacted because of the load-/store expansion with UPC and Chapel shared memory accesses is not explored.

## 6. CONCLUSION

The LLVM PIR proposed in this paper can leverage parallel optimization of parallel languages and libraries by focusing on structured parallel constructs, which are easier to optimize, while minimizing the number of built-ins and expressing as much parallelism as possible. This extension to LLVM IR includes (1) a parallel execution attribute for each group of statements, (2) a high-level synchronization attribute on each statement and an API for low-level synchronization events, and (3) data location on processes together with two built-ins for implementing communications in message-passing memory systems. We have implemented a part of this parallel extension into the LLVM compiler and tested two possible resulting OpenSHMEM one-sided communication optimizations.

Besides the loop tiling and communication vectorization passes addressed in this paper, future work will look at applying and adapting other transformations performed by LLVM to OpenSHMEM benchmarks and applications.

## 7. REFERENCES


[1] HPCTools PGAS-Microbenchmarks. `https://github.com/uhhpctools/pgas-microbench`.

[2] The X10 programming language. `http://www.research.ibm.com/x10`.

[3] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl*, 21:291–312, Aug. 2007.

[4] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, 2010.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[6] E. W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[7] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[8] A. Hayashi, R. Surendran, J. Zhao, M. Ferguson, and V. Sarkar. LLVM Optimizations for PGAS Programs - Case

Study: LLVM Wide Optimization in Chapel -. In *The 1st Chapel Implementers and Users Workshop (CHIUW) (co-located with IPDPS2014)*, Phoenix, AZ, USA, 2014.

[9] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD Distributed-memory Machines. *Commun. ACM*, 35(8):66–80, Aug. 1992.

[10] J. Jose, K. Kandalla, M. Luo, and D. Panda. Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In *41st International Conference on Parallel Processing (ICPP)*, Sept. 2012.

[11] P. Jouvelot and R. Triolet. Newgen: A Language Independent Program Generator. Technical report, CRI/A-191, MINES ParisTech, Jul. 1989.

[12] D. Khaldi, P. Jouvelot, F. Irigoin, and C. Ancourt. SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension. In *Local proceedings of the 17th Workshop on Compilers for Parallel Computing*, CPC'13, 2013.

[13] D. Khaldi, P. Jouvelot, F. Irigoin, and C. Ancourt. The SPIRE Methodology of Compiler Parallel Intermediate Representation Design. Technical report, CRI/A-613, MINES ParisTech, `http://www.cri.ensmp.fr/classement/doc/A-613.pdf`, Feb. 2015.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[15] The LLVM Development Team. *LLVM Language Reference Manual (Version 3.8)*, 2015.

[16] J. Merrill. GENERIC and GIMPLE: a New Tree Representation for Entire Functions. In *GCC Developers Summit*, pages 171–180, 2003.

[17] D. Novillo. OpenMP and Automatic Parallelization in GCC. In *the Proceedings of the GCC Developers Summit*, Jun 2006.

[18] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17:1–31, Aug. 1998.

[19] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A Unified Deadlock-Free Construct for Collective and Point-To-Point Synchronization. In *ICS'08*, pages 277–288, New York, NY, USA, 2008. ACM.

[20] S. Watanabe, G. Funck, and N. Vukicevic. Clang UPC - UPC pointer-to-shared in LLVM IR. Technical report, Intrepid Technology, Inc., 2014.

[21] J. Zhao and V. Sarkar. Intermediate Language Extensions for Parallelism. In *Proc. of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 329–340, New York, NY, USA, 2011. ACM.